7-1 指標(pointer)

記憶體-位址

當宣告一個變數並賦予它初值後，我們可以確定這個值一定存放在電腦記憶體的某個地方，問題是它到底放在哪裡呢？在地球表面上我們可以用經緯度來標定一個位置，而在電腦裡要標定記憶體中的某個位置則是要靠「位址(address)」



我們在寫程式時可以用 `cout << a` 來印出變數 a 的值，但大家必須了解背後的實際動作是將儲存 a 的那塊記憶體內容印出來。

眼尖的同學應該注意到了上圖中的位址每一個相差 4，這是因為我們以 int 型別的變數為例，而 int 的大小是 4 byte，所以每個 int 都要在記憶體中佔掉 4 byte 的空間。若是我們使用 double 型別，則每個變數都會佔掉 8 byte 的空間。

由於每次程式載入記憶體執行時可能都在不同的位置，因此這次變數 a 儲存在 0x22ff18 不表示下次執行時它也會儲存在 0x22ff18。使用取址(address-of)運算子 &可以取得變數目前在記憶體中的位址。

```
int a = 2, b = 3;
cout << &a << endl;
cout << &b << endl;
```

0x22ff18 0x22ff14

指標變數

在 C/C++ 中用來儲存位址的是一種特殊型別的變數－指標(pointer)變數

宣告

```
資料型別 *變數名稱; // 注意前面有個 * 號
```

範例

```
int *pNumber;         // 宣告一個名為 pNumber 的指標變數，用來指向一個 int 型別的變數

float *pF = nullptr; // 宣告一個名為 pF 的指標變數，用來指向一個 float 型別的變數，
                     //並給定指標的初值為 nullptr，即不指向任何地方的空指標。
```

取址(address-of)運算子 &

在變數名稱前加上一個取址運算子(&)可以取得該變數的位址。

```
int a = 6;
int *pA = nullptr;
pA = &a; // 取得變數 a 的位址並儲存在指標 pA 中
```

提領(dereference)運算子 *

在指標變數名稱前加上一個提領運算子 *， 可以 讀/寫 它所指向變數的值。

```
int a = 6, b = 5;
int *pNum = nullptr;

pNum = &a;
cout << *pNum << endl;
pNum = &b;
cout << *pNum << endl;
```

6 5

```
int a = 6, b = 5;
int *pNum = nullptr;

cout << "a = " << a << ", b = " << b << endl;
pNum = &a;
*pNum = 5;
pNum = &b;
*pNum = 6;
cout << "a = " << a << ", b = " << b << endl;
```

a = 6, b = 5 a = 5, b = 6

```
int a = 6, b = 5;
int *pNum1 = nullptr;
```

```
int *pNum2 = nullptr;

cout << "a = " << a << ", b = " << b << endl;

pNum1 = &a;
pNum2 = pNum1;
*pNum2 = 3;

cout << "a = " << a << ", b = " << b << endl;
```

a = 6, b = 5 a = 3, b = 5

動態配置記憶體

截至目前為止，我們的程式都在一開始就將需要使用的記憶體（如：變數、陣列）大小寫死在程式碼中。

```
int a = 0, b = 0; // 兩個 int 變數，共 2*4=8 byte
int score[50];    // 一個包含 50 個 int 的陣列，共 50*4=200 byte
```

但是有時候我們在寫程式時並不知道使用者執行時需要多大的空間。例如我們要寫一個讀入學生成績並依成績高低排序的程式，你可能會想這樣寫：

```
int numOfStd=0;
int score[50];

cout << "請輸入學生人數：";
cin >> numOfStd;

for(int i=0; i<numOfStd; i++) {
    cin >> score[i];
}
// 排序
......
```

宣告 50 個整數大小的陣列來存於成績似乎是個合理的作法，因為目前高中以下的每班人數多不超過 50 人，但.....要是超過了怎麼辦？那就設成 100吧！要是人家拿來做全校學生的排序怎麼辦？那改成 10000吧！這是個大問題，因為設大了浪費，設小了又無法運作。

使用 C99 的可變長度陣列是個方法。

```
int numOfStd=0;
cout << "請輸入學生人數：";
cin >> numOfStd;

int score[numOfStd];  // C99 的可變長度陣列
```

```
for(int i=0; i<numOfStd; i++) {
 cin >> score[i];
}......
```

但它不是 C++ 標準裡的必要特性，不是所有的 C++ 編譯器都支援，而且只能在函數內部使用，無法放在全域區，再者使用到的是堆疊記憶體，大小較受限。

為了解決前述的兩難狀況，我們必須有一個能在程式執行間動態依需求配置記憶體的方法。

在 C++ 中，我們可以用 new 這個關鍵字來要求配置一定大小的記憶體，若是成功要到指定大小的記憶體，它會回傳這塊記憶體的開頭位址，我們可用指標把它存起來。

配置

```
new 資料型別;       // 配置單一變數
new 資料型別[數量]; // 以陣列方式配置
```

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];
......
```

存取

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];

for(int i=0; i<numOfStd; i++) {
    cin >> score[i];
}
// 排序
......
```

也可以這麼做

```
int numOfStd=0;
int *score;

cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];

for(int i=0; i<numOfStd; i++) {
    cin >> *(score+i);
}
// 排序
......
```

釋回

當不在需要使用到先前配置的記憶體時，記得要用 delete 將記憶體還給系統，讓其他程式可以使用該記憶體。

```
delete 指標名稱;      // 釋回單一變數所配置記憶體

delete [] 指標名稱;  // 釋回陣列所配置記憶體
```

```
int numOfStd=0;
int *score;

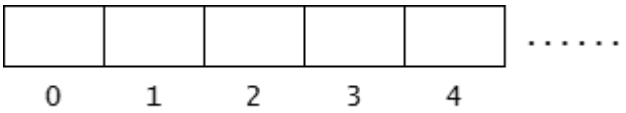
cout << "請輸入學生人數：";
cin >> numOfStd;

score = new int[numOfStd];

for(int i=0; i<numOfStd; i++) {
 cin >> score[i];
}
// 排序
......
delete [] score;
```

位址空間

在電腦裡面儲存資料的最基本單位是位元(bit)。而在記憶體中，我們存取資料的基本單位則是位元組(Byte)。我們可以把電腦的記憶體想像成是一連串的小盒子，每一個小盒子裡面可以放 1 Byte 的資料，這些盒子被按照順序加以編號，這個編號我們稱之為「位址」。



我們若是用 4 Byte 來儲存位址，則編號的範圍也就是位址的範圍可由 00 00 00 00 到 FF FF FF FF，共有 $ 2^{32} $ Byte，也就是 4 GB 的空間。

這就是為什麼 32 位元的電腦和作業系統無法存取超過 4 GB 記憶體的原因。64位元的系統用 8 Byte 來儲存位址，他可以定址的範圍為 00 00 00 00 00 00 00 00 到 FF FF FF FF FF FF FF FF，共有 $ 2^{64} $ Byte，即 $ 2^{34} $ GB 的空間。

Code::Blocks 近期的版本預設安裝 64 位元的編譯器，所以編譯出來的程式是 64 位元的。用下面這段程式碼可以看到整數的指標是 8 Byte。

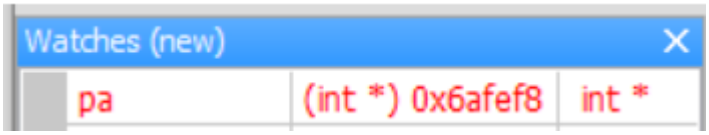
```
cout << sizeof(int*) << endl;
```

觀察資料在記憶體中的存放

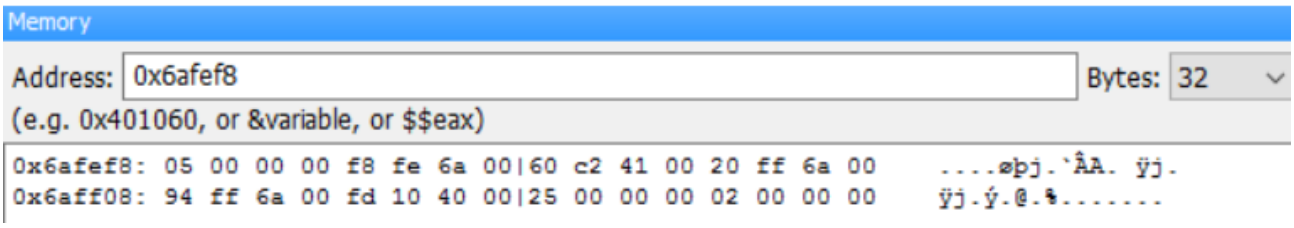
接下來我們使用 code::blocks 的 memory 視窗來觀察資料在記憶體中的存放方式。我們在下面這段程式的第 05 列設一個中斷點，用 debug 模式執行到該處。

```
int main()
{
        int a = 5;
        int *pa = &a; // &a 表示取得變數 a 在記憶體中的位址
        cout << a << endl;
        return 0;
}
```

用 watch 觀察 pa 的值，我們可以看到儲存 a 的記憶體位址，你看到的值可能有所不同，這是因為每次程式載入時會在記憶體的哪個地方是不一定的。



點選 [Debug]→[Debugginh windows] →[Memory dump]，打開記憶體檢視視窗。在Address 的地方輸入 pa 的值，從 0x6afef8 開始的連續4個 Byte 就是 a 的值。



你可能會覺得得怪為什麼是 05 00 00 00，而不是 00 00 00 05，這個和 Intel、AMD 的 CPU 設計有關，它會剛好反過來存放。

接下來我們觀察一下陣列中的資料存放方式。在這裡我們要觀察的重點是：

- 陣列中的資料是緊臨儲存在一起的
- 在程式碼中陣列的名稱就等於指向第一個元素的指標
- 指標加 1 後，值會變成多少？

```
int main()
{
        int a[5] = {100,200,400,600,800};
        int *pa = &a[0];
        pa = pa+1;
        pa = pa+1;
        pa = pa+1;
        pa = pa+1;
        return 0;
}
```

陣列名稱 a 其實就是 a[0] 的位址，也就是陣列在記憶體中的開始位址。在上面那段程式的第 05 行之後，使用 pa[0], pa[1], pa[2], ……，就等於 a[0], a[1], a[2], ……。

觀察傳值與傳址呼叫

接下來我們藉著 Memory dump 視窗來觀察傳值呼叫與傳址呼叫的行為。

```
void f(int a)
{
        a = 5;
        return;
}

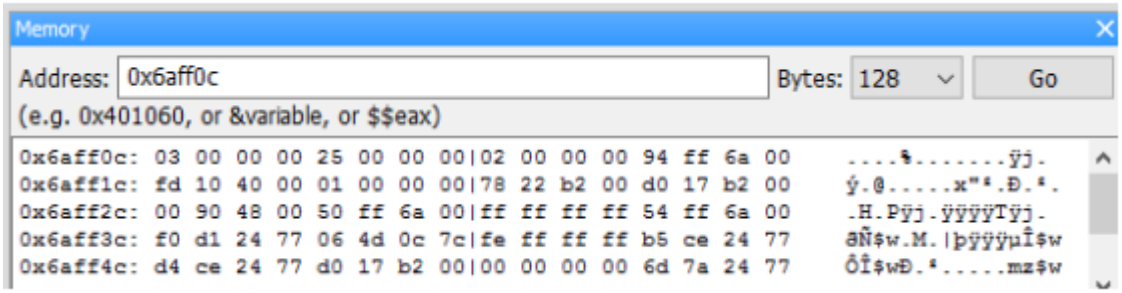
void g(int *a)
{
        *a = 5;
        return;
}

int main()
{
        int a = 3;
        f(a);
        g(&a);
        return 0;
}
```

將中斷點設在 16 列的地方。執行到中斷點時，把 &a 加入 watch。

&a	(int *) 0x6aff0c	int *

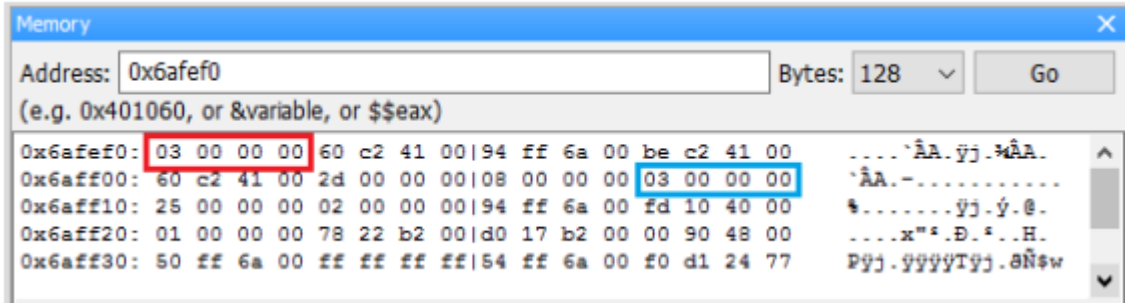
接著到 Memory dump 中找到 0x6aff0c 的地方，確認 03 00 00 00 在那裡。



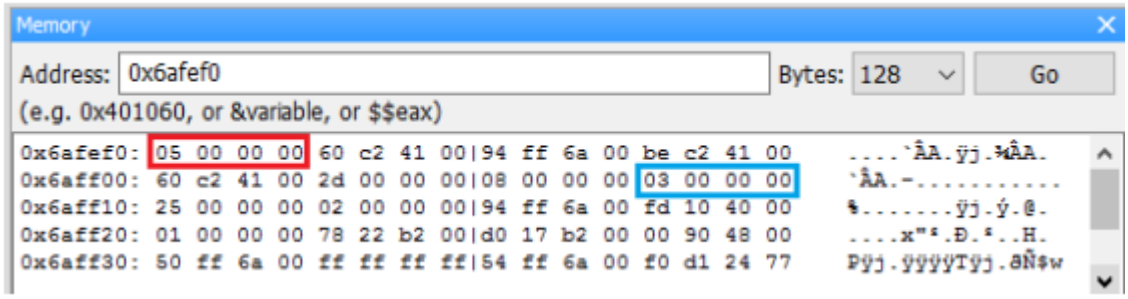
用 step into 追蹤到函數 f 裡面，這時你會發現在 watch 中的 &a 變了，因為這裡的 a 是函數 f 裡的區域變數 a，他的位址是 0x6afef0。

&a	(int *) 0x6afef0	int *

從這裡可以看到 f(3) 裡的參數 3 被複製到區域變數 a 裡，右下角 0x6affc0 是 main 函數裡的區域變數 a 。

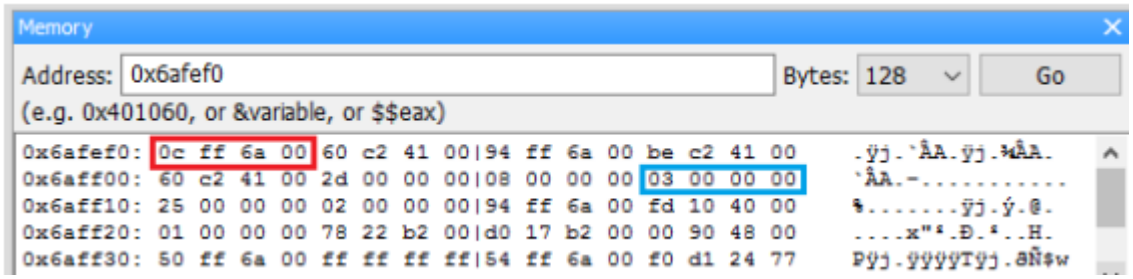


用 Next Line 執行下一行 “ a = 5; ”，可以看到函數 f 裡的區域變數 a (0xfafef0) 變成 5，而main 函數裡的區域變數 a (0xfaff0c) 值不變，依然還是 3。

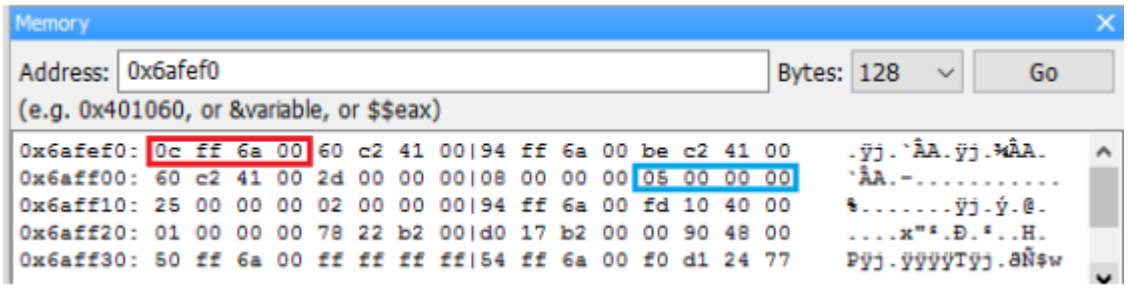


繼續用 Next Line 執行，直到返回 main 裡面。這時 watch 裡 &a 的值又變回 0xfaff0c，表示這時看到的 a 是 main 函數的區域變數 a。

比照剛才的做法，用 Step into 蹤至函數 g 裡，很巧的這次函數 g 裡的區域變數 a，放在0x6afef0。不過它的值不是3喔，我們傳入 g 的是 main 函數裡區域變數 a 的位址，所以在Memory dump 中，可以看到區域變數的值是 0c ff 6a 00，即 main 函數中區域變數 a 的位 址。



用 Next Line 執行 “ *a=5; ” 這行，可以看到 0x006aff0c 這個位址的資料由 03 00 00 00 變成了 05 00 00 00。



這是因為我們使用 * 運算子，操作 a 所儲存位址(0x6aff0c)內的值。

繼續用 Next Line 執行，直到返回 main 裡面。這時 watch 裡 &a 的值又變回 0xfaff0c，表示這時看到的 a 是 main 函數的區域變數 a。

至此我們可以觀察到以下幾點：

1. 不管是傳值呼叫的 f，亦或是傳呼叫的 g，都是把呼叫函數時的引數值複製到被呼叫函數的參數(也是它的區域變數)中。只是前者複製的是變數的值，後者複製的是變數在記憶體中的位址。
2. 相對於傳入變數的值，傳入變數的位址讓我們可以直接藉由編輯該位址記憶體的值，達到修改外界變數值的目的。

Revision #9
★Created 2026-03-29 23:43:52 UTC by huihui
Updated 2026-04-13 10:40:57 UTC by huihui